Parveen Khurana

125 Followers About Following

Encoder-Decoder Model for image Captioning, Machine Translation and Machine Transliteration:

P Parveen Khurana Jul 26, 2019 · 14 min read

This article covers the content discussed in the Encoder-Decoder Models module of the Deep Learning course offered on the website: https://padhai.onefourthlabs.in

Encoder Decoder Model for Image Captioning:

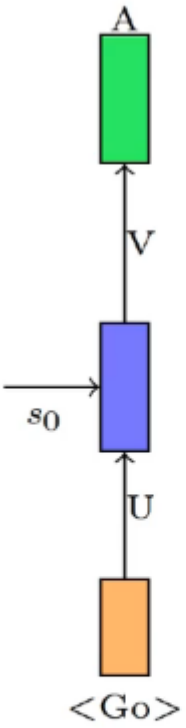
So, far we have discussed how to generate the next word given a previous set of words.

- So far we have seen how to model the conditional probability distribution $P(y_t|y_1^{t-1})$
- More informally, we have seen how to generate a sentence given previous words
- What if we want to generate a sentence given an image?

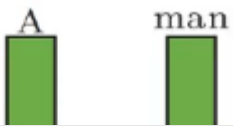
A man throwing
a frisbee in a park

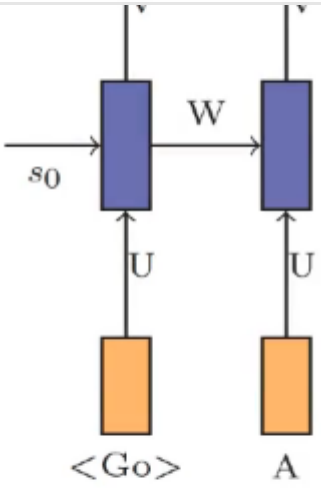
Given an image, we want to generate a suitable caption

Input, in this case, is the image and output should be a suitable description of the input image. And so the model should do this one word at a time as depicted below:

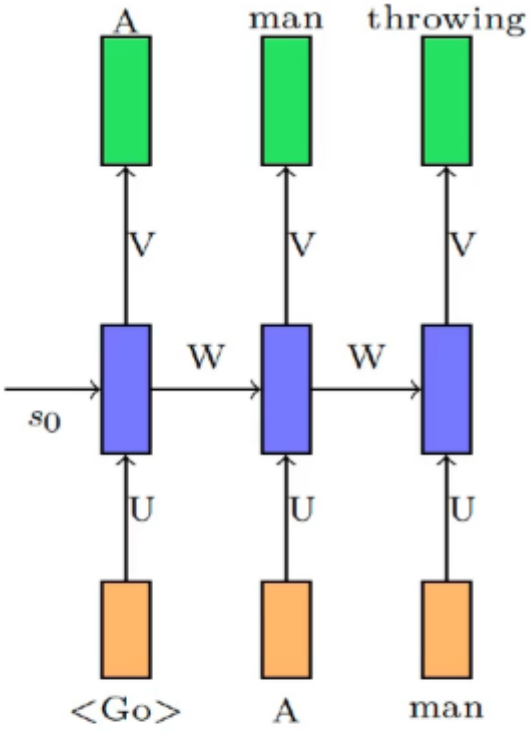


A man throwing
a frisbee in a park

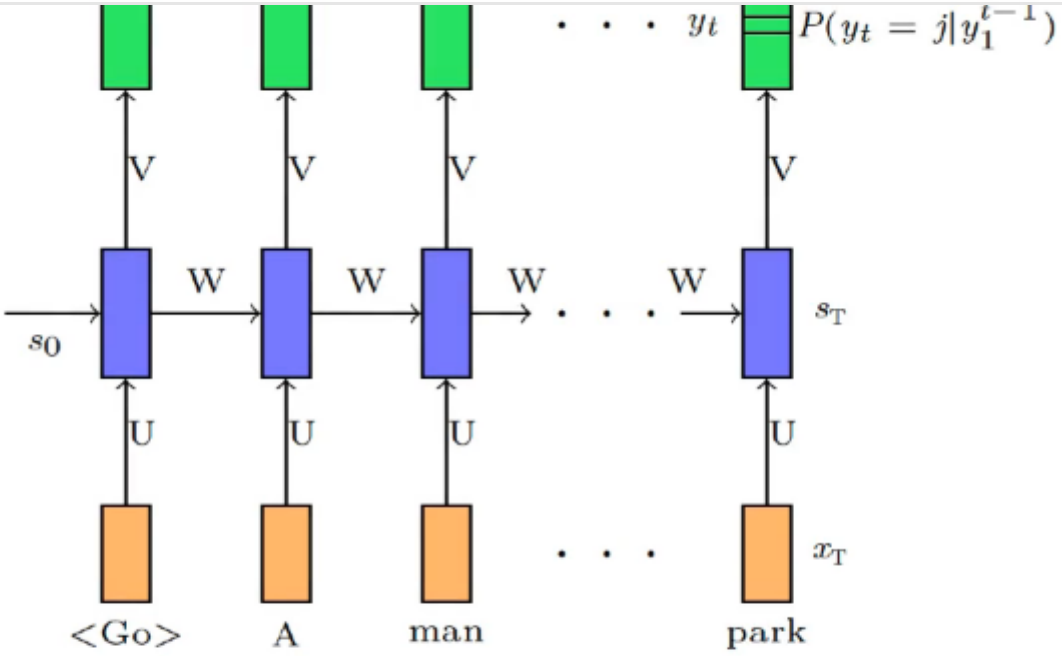




A man throwing
a frisbee in a park



A man throwing
a frisbee in a park



A man throwing
a frisbee in a park

So, the way it would work is that take in the image input and generate the first word and now based on the image and the first word, we generate the second word; then based on the first two words and the image, we generate the third word and so on.

We have seen how to generate the third word given the first two words in the previous example of Language Modelling(https://medium.com/@prvnk10/encoder-decoder-models-fa0f155cf042), so we have added one more complexity to the task that not just the previous words but the image also is used to generate the next word.

Now just as earlier we were writing task as predicting the t'th word given the previous words as

$$P(y_t|y_1^{t-1})$$

$$P(y_t|y_1^{t-1}, I)$$

i.e given the previous words and the image, predict the t'th word.

- We are now interested in $P(y_t|y_1^{t-1}, I)$ instead of $P(y_t|y_1^{t-1})$ where I is an image
- Notice that $P(y_t|y_1^{t-1}, I)$ is again a conditional distribution

Earlier we got rid of the terms from **y1, y2, ……, y(t-1)** by encoding all of the them in a single representation which was the RNN state vector **st** at the **t'th** time step because it depends on the input of all the previous time steps, then we computed the **final output distribution** as a function of this state **st** as

$$\hat{y} = O(Vs_t + c)$$

- Earlier we modeled $P(y_t|y_1^{t-1})$ as

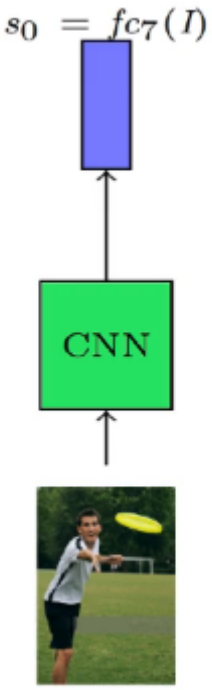
$$P(y_t|y_1^{t-1}) = P(y_t = j|s_t)$$

- Where s_t was a state capturing all the previous words

image into a vector and we will use a simple CNN for this encoding of the image.

- We could now model $P(y_t = j|y_1^{t-1}, I)$ as $P(y_t = j|s_t, f_{c_7}(I))$
- where $fc_7(I)$ is the representation obtained from the fc_7 layer of an image

Let's say we pass the image through VGG-16, say at the output layer just before the last fully connected layer, we take whatever representation that we get. We could have chosen to take the image representation from any of the layers we wish to but it is observed that the output taken from deeper layers is always better. So, we use this as the representation of the image.

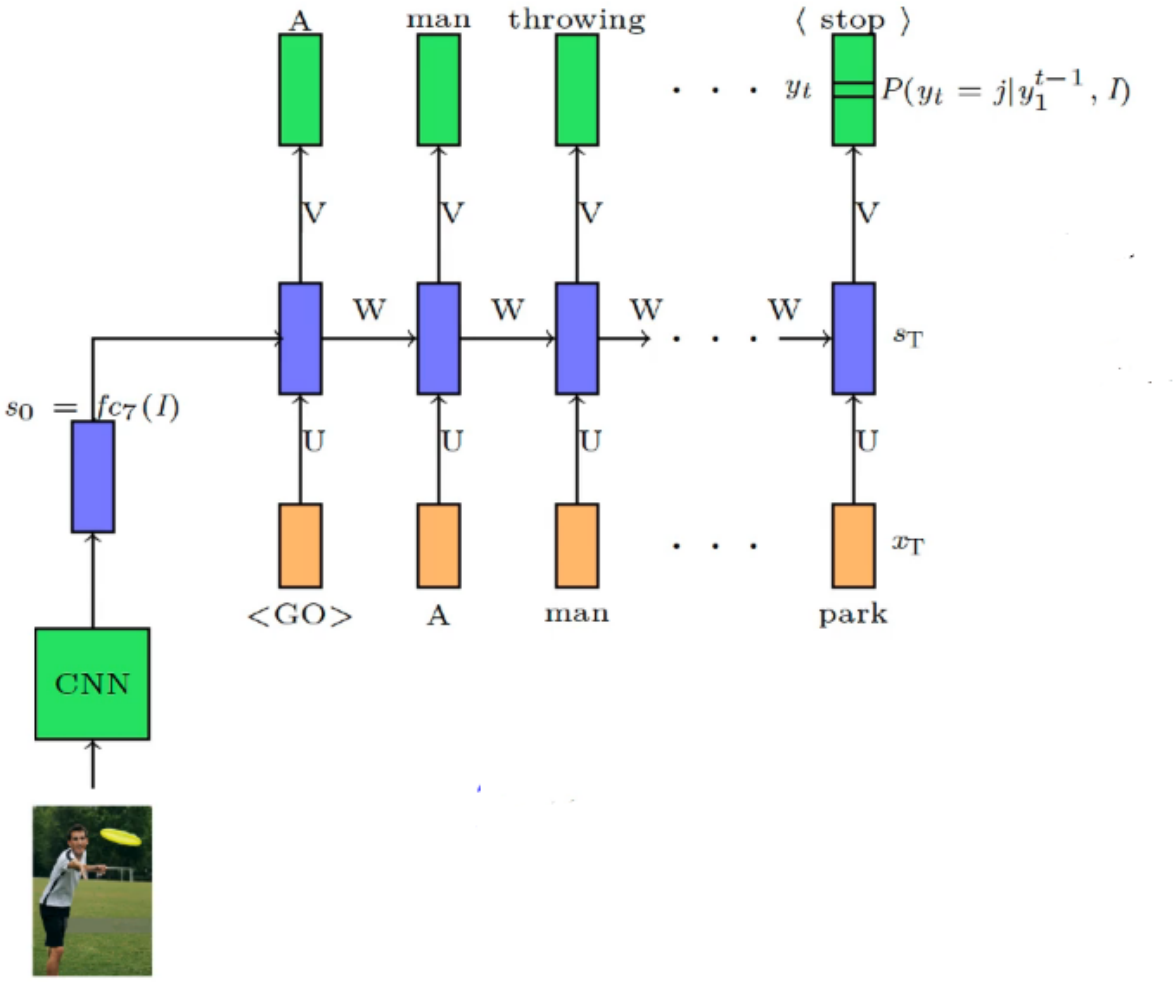


Now the final output **y** should be a function of **st**(which is a compact representation of all the words generated so far), and the representation of the image fc7(I); both of

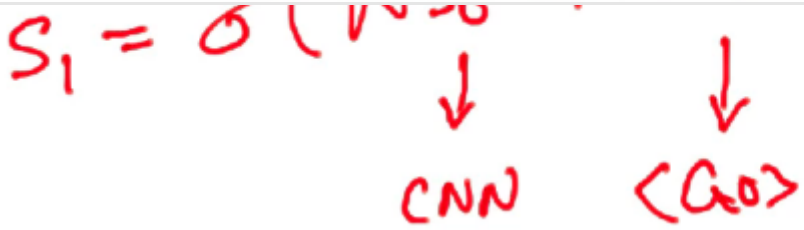
- There are many ways of making $P(y_t = j)$ conditional on $f_{c_7}(I)$
- Let us see two such options

Option 1:

- **Option 1:** Set $s_0 = f_{c_7}(I)$
- Now s_0 and hence all subsequent s_t's depend on $f_{c_7}(I)$



Now we compute s1 as:



'so' is coming from CNN, 'x1' is one hot encoded version of <GO>

s1 would have the encoded information of the image and the input <GO> and we use this to predict the output.

Now **s2** is dependent on **s1** and **s1**, in turn, depends on **s0** and **s0** in turn depends on the image, so we can say that the state vector flowing through the network has some information from the image. In this case, one issue is there that if many time steps are there, then the information would get morphed but this is one way of making every output dependent on the image as every output would depend on the state and we have encoded the image information in the state at the very beginning(**s0**).

- **Option 1:** Set $s_0 = f_{c_7}(I)$
- Now s_0 and hence all subsequent s_t's depend on $f_{c_7}(I)$
- We can thus say that $P(y_t = j)$ depends on $f_{c_7}(I)$.
- In other words, we are computing $P(y_t = j | s_t, f_{c_7}(I))$

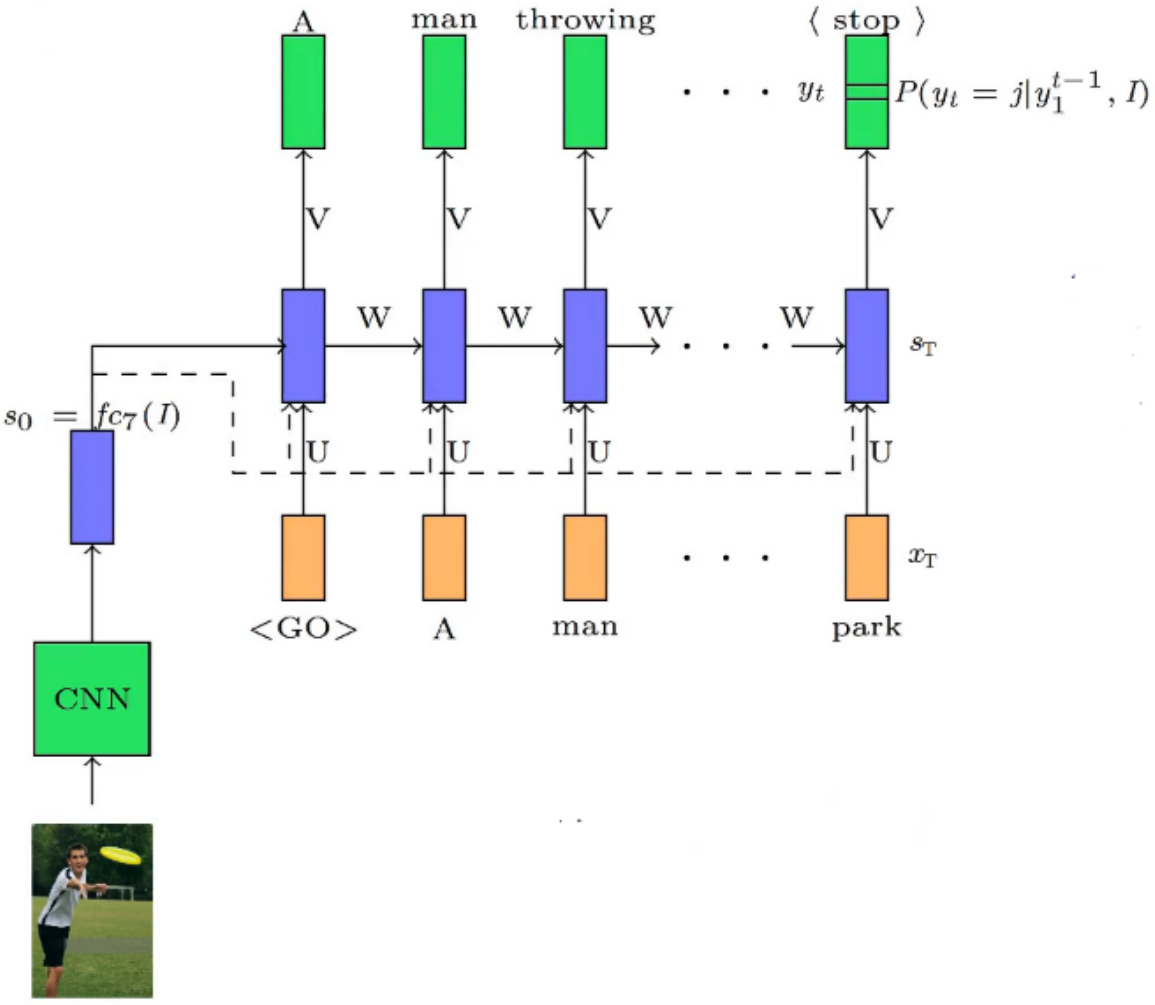
Option 2:

The other option is to feed the output of the CNN explicitly at every time step.

- **Option 2:** Another more explicit

$$s_t = RNN(s_{t-1}, [x_t, f_{c_7}(I)])$$

- In other words we are explicitly using $f_{c_7}(I)$ to compute s_t and hence $P(y_t = j)$



We compute s2 in this case as:

$$s_t = f(W s_{t-1} + U[x_t, s_0] + b)$$

[x1, s0] means the concatenation of these two vectors

The issue that we had with the first option(image information getting morphed in many time steps are there) is taken care of by using this option.

Now we compute y2 as

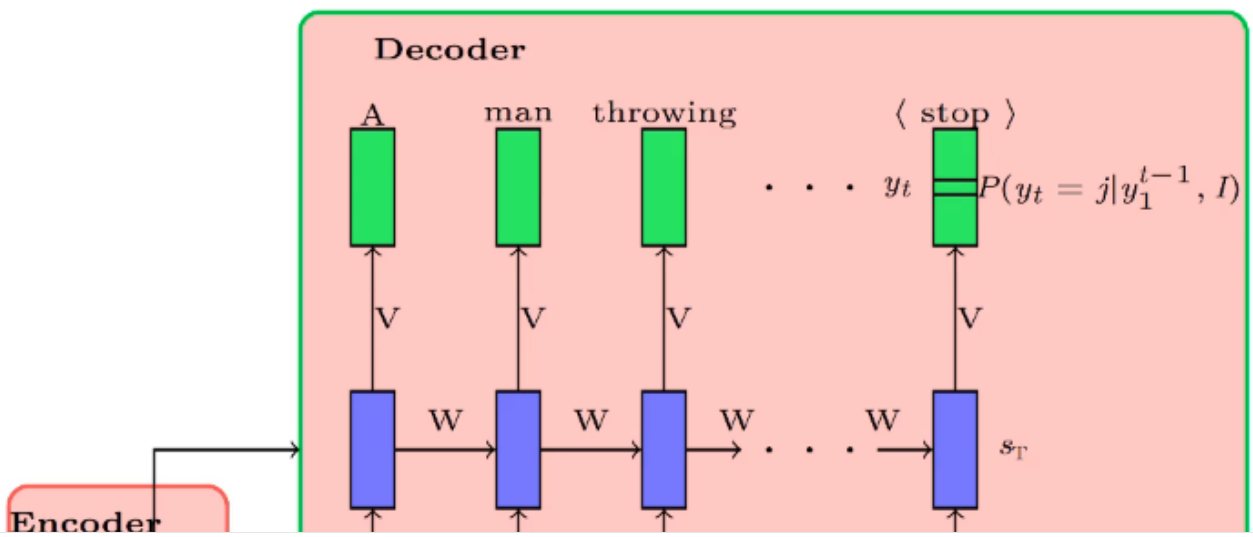
y2 = O(V*s2 + c)

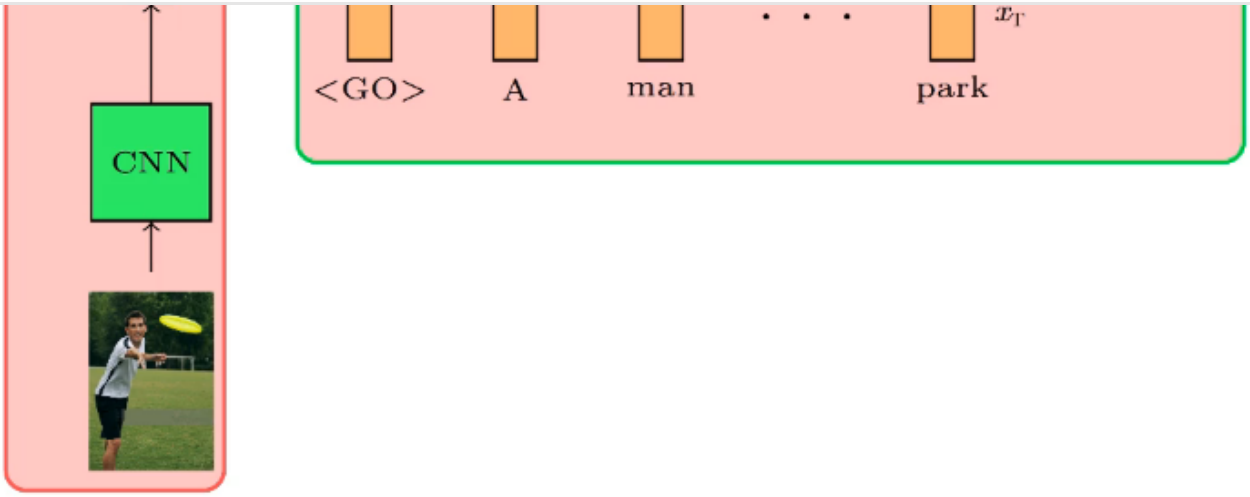
so **y2** is dependent on **s2** and **s2**, in turn, depends on **s1**, **x1,** and **s0** where **s0** is actually the encoded form of the image.

Now we can say that **y2** is conditional not only on **y1** but also on **I**(Image) and that's exactly what we wanted to capture.

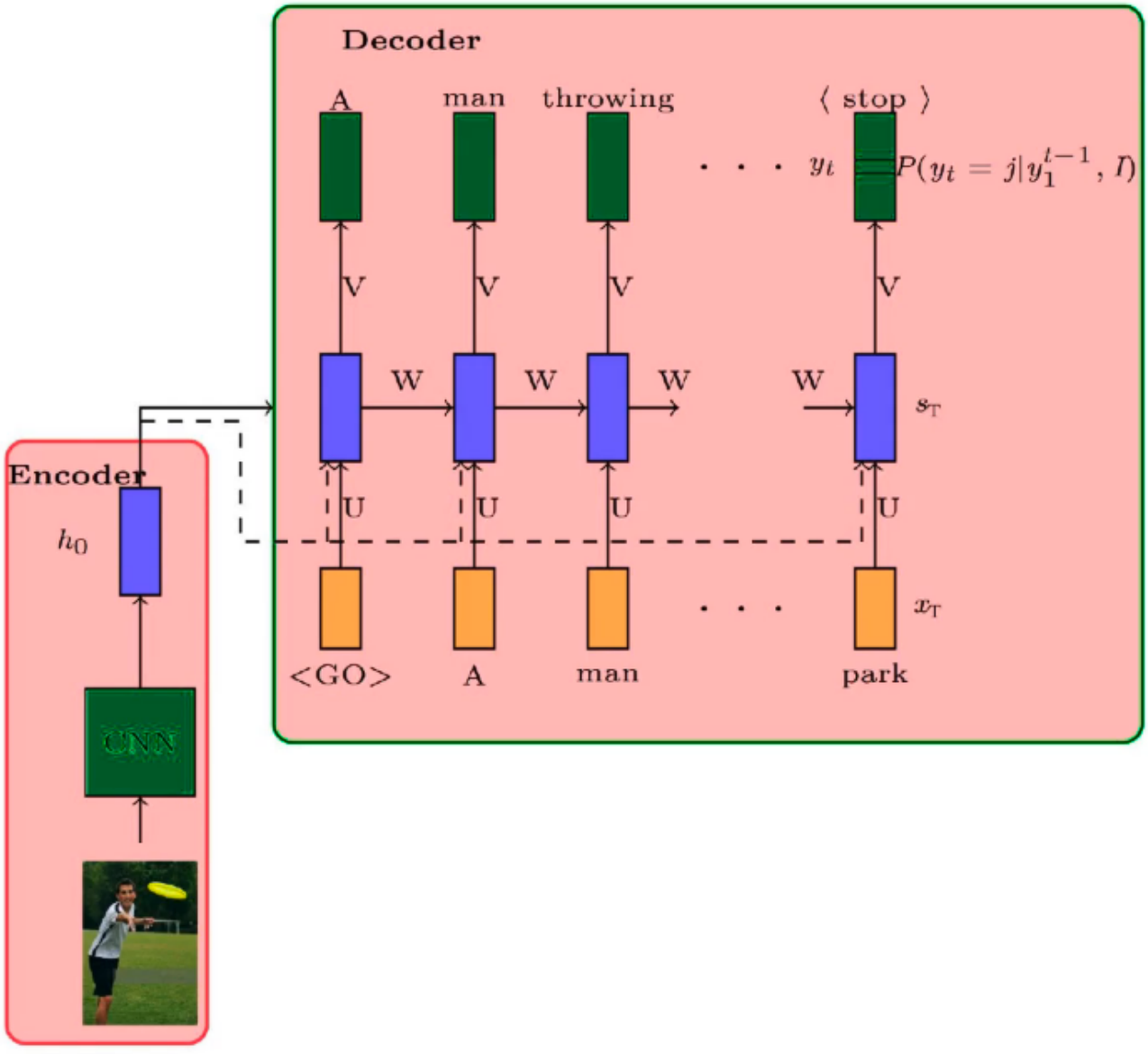
Full Architecture:

We have an image and when we pass it through CNN, we get the encoded form of the image, so CNN is the encoder which encodes the input(input is an image, we are given an image and we are asked to generate the caption for the same, so the input is an image only). Now the decoder is a combination of RNN and single-layer feed-forward neural network(input to this feed-forward network is the state vector and the output is the distribution coming out of the softmax function), which takes this encoded image either at time step 0 or this encoded image is being fed at every time step and then its decoding the encoded information one word at a time to generate the output.





Architecture 1



Architecture 2

architecture

- A CNN is first used to **encode** the image
- A RNN is then used to decode (generate) a sentence from the encoding
- This is a typical **encoder decoder architecture**
- Both the encoder and decoder use a neural network

Six Jars for Image Captioning:

Task: Image Captioning

Our **x** would be a single image as the input and the output **y** could be any number of words so the output is a sequential output.

- **Task:** Image captioning
- **Data:** $\{x_i = image_i,\ y_i = caption_i\}_{i=1}^{N}$

Data:

The data, in this case, is a number of images with their correct description/caption.

Model:

We want the output as a function of the input and we produce output at every time step so we write as:

- **Model:**
  - **Encoder:**

$$s_0 = CNN(x_i)$$

  - **Decoder:**

$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$

$$P(y_t|y_1^{t-1}, I) = softmax(Vs_t + b)$$

e(y_hat(t-1)) represents the one-hot encoded form of the output at (t-1)th time step

We pass the input **xi**(here 'i' is not for the time step, it represents the i'th training example) through the CNN which encodes the information, then we pass the encoded information of this image and the one-hot encoded information of the previous output through the RNN at every time step which gives the value of the state at that time step and using the state we compute the final distribution.

Parameters: would include all the parameters of the RNN, CNN, biases.

- **Parameters:** $U_{dec},\ V,\ W_{dec},\ W_{conv}, b$

Loss:

- **Loss:**

$$\mathscr{L}(\theta) = \sum_{i=1}^{T} \mathscr{L}_t(\theta) \quad = -\sum_{t=1}^{T} \log P(y_t = \ell_t | y_1^{t-1}, I)$$

Once we have the loss we can update all the parameters as per the update rule of the Algorithm.

Encoder-Decoder for Machine Translation:

Here our task is Machine Translation where the input is a sequence and the output is also a sequence and the length of input and output sequence could be different. This can not be modeled as a Sequence Labeling Problem as the order of words(in the example below, see the first word in both sentences mean the same but the **second word in the input** which is '**am**' when translated to Hindi is placed at the **end of the sentence** as the word '**Hoon**') in different languages could be different example:

i/p : I am going home

o/p : Mein ghar ja raha hoon

And if we produce output at every time step(i.e for every input) then we would get only 4 words at the output(as input length is 4) but actually we need to produce 5 words(in the above case) and the length of output in the above example is 5 whereas the input length is 4. So, this can not be modeled as Sequence Labeling Problem and of course, this is not a Sequence Classification Problem as well because we have to produce multiple outputs and not just a single output. So, this falls into the Sequence to Sequence generation problems.

Now from the Encoder point of view, we get a sentence as the input and we encode the input and from this encoded input, we start producing the output in Hindi.

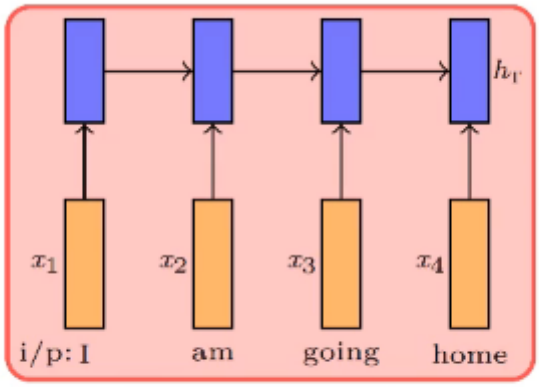
Data:

So, the input is a sequence and the output is also a sequence and we have this pair of sequences and we would be given many such sequences say 'N' where 'N' is very large and in fact for training state of the art of machine translation systems using neural networks we need order of Billions of such parallel sentences.

Model:

Option 1:

Since the input is a sequence, we use RNN as the encoder. We represent each word in the form of one hot vector and we feed this one hot vector to the RNN.



i/p : I am going home

We compute **h1** (the hidden state at **time step 1**) as

$$h_1 = \sigma(Wh_0 + Ux_1 + b)$$

where we assume for now that **W, U, b, h0** are given to us.

x1 is the one hot representation of the word '**I**'(first word in the input)

Once we have **h1**, we repeat the same story for **h2, h3, h4**.

The last representation that we would have is **ht**(where **t** is the number of input words in the input sequence which is 4 in this case). And this last representation would have captured the information of all the inputs, so we can think of this last representation as the final encoding of the input sentence which was given.

$$h_t = RNN(h_{t-1}, x_{it})$$

Now at the decoder, we need to generate one output at a time and its again a language modeling task and the way we would write it is:

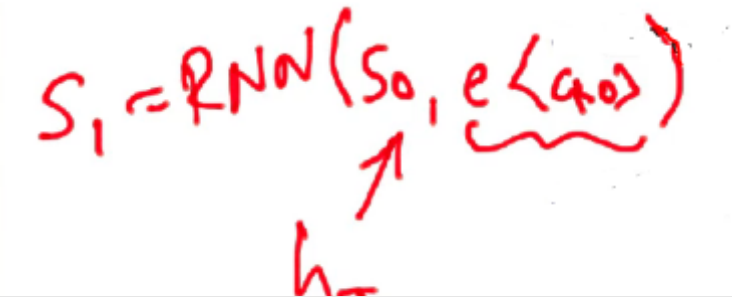
$$P(y_t|y_1^{t-1}, x) = softmax(Vs_t + b)$$

We want to generate the next word based on all the words that we have generated so far and also based on the original input which was given to us. And we approximated **y1, y2, … y(t-1)** as **st**(decoder hidden state). And similarly the input **x** we can approximate it by the final encoded vector for it which is **ht** which we can also call as **s0** for the decoder.

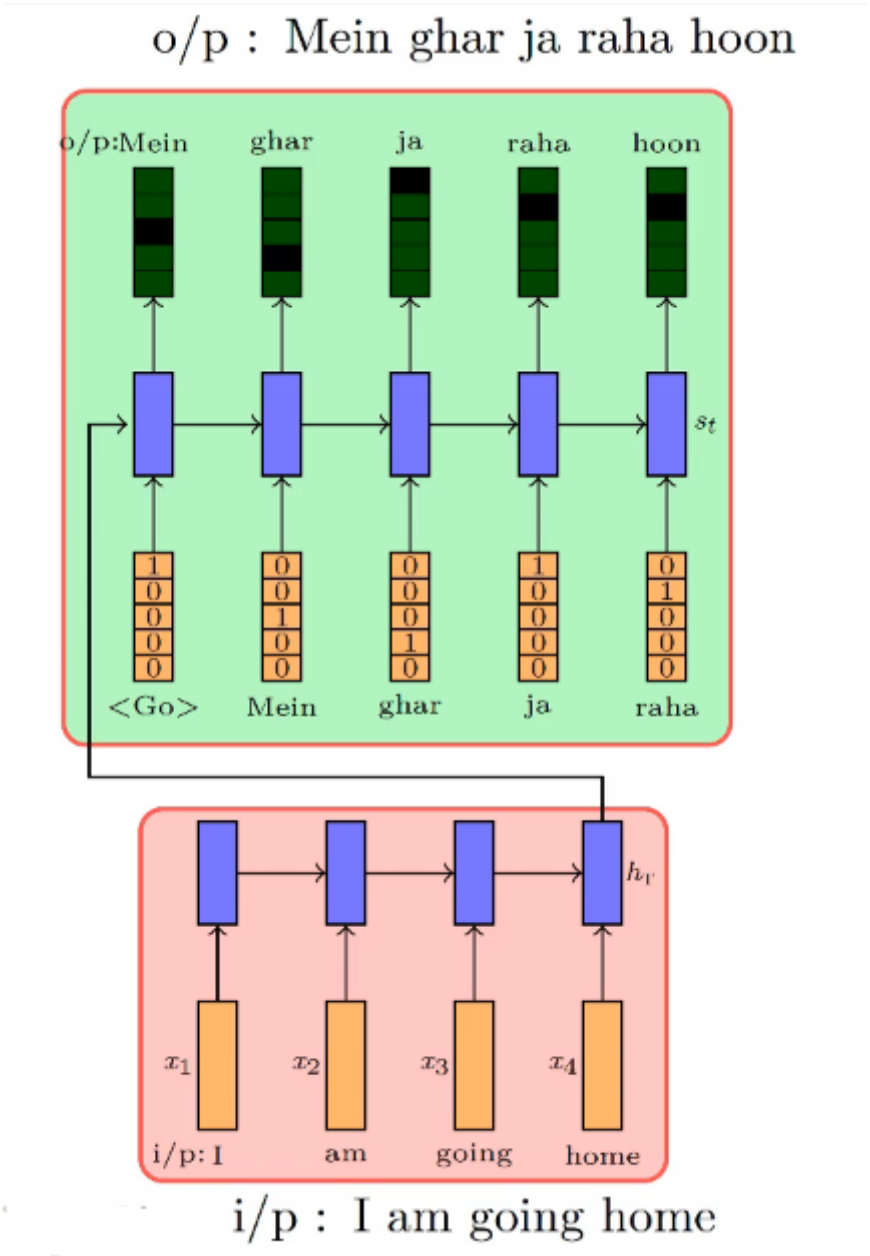
- **Decoder:**

$$s_0 = h_T \quad (T \text{ is length of input})$$
$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$
$$P(y_t|y_1^{t-1}, x) = softmax(Vs_t + b)$$

So, we are feeding **ht** as **s0**(**ht** has been fed to the 0'th time step of the decoder RNN) and then the decoder RNN at every time step is generating a hidden state based on the previous hidden state as well as based on the one-hot encoding of whatever we had generated before and now we compute **s1** as:

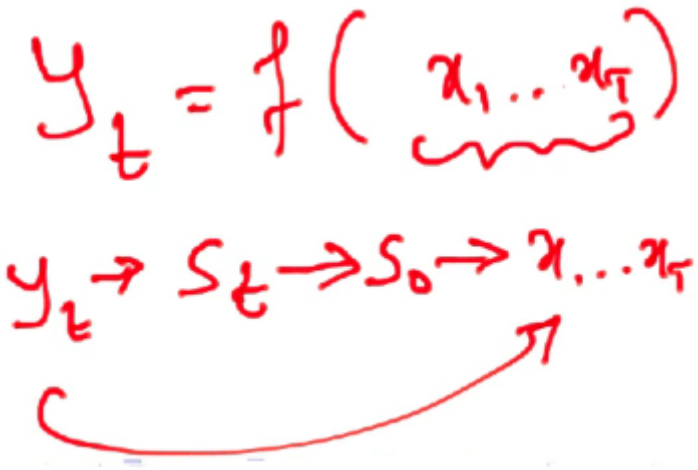


And based on this **s1** we want to generate a probability distribution over the vocabulary of the Hindi words. So, this gives us a probability distribution and we also know the true distribution from where we go towards the loss.



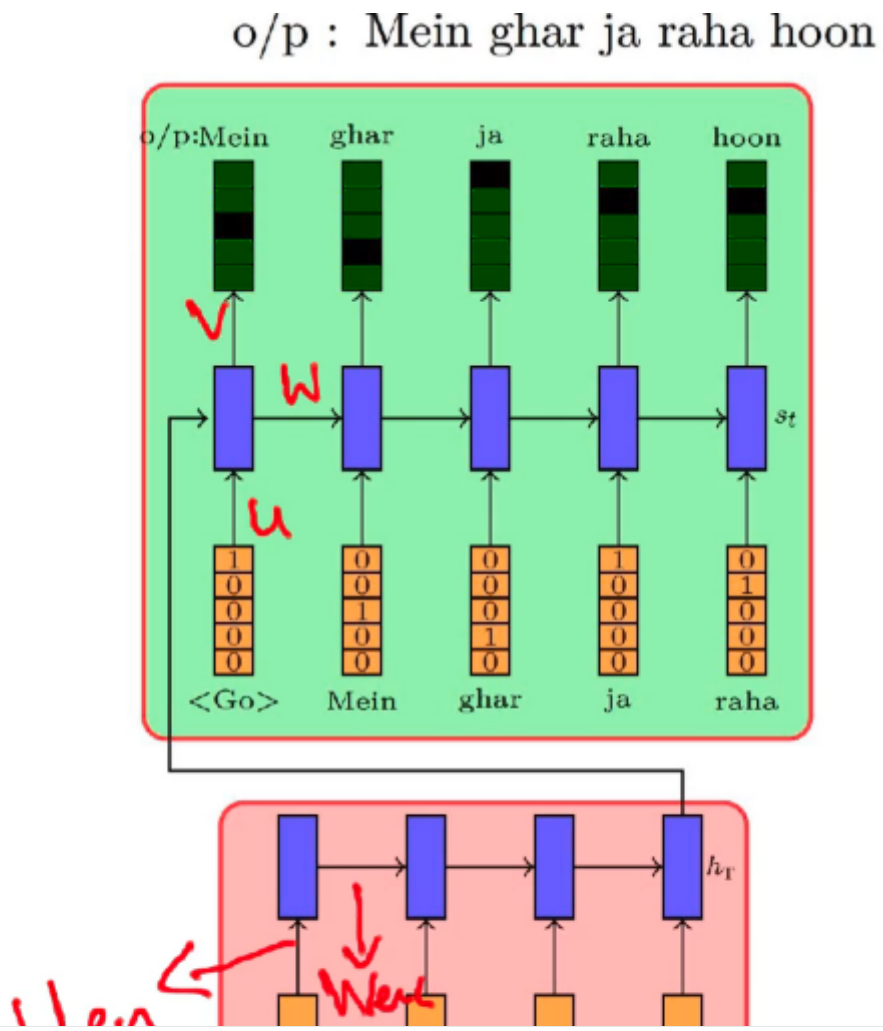
So, the output has been represented as the function of the input.

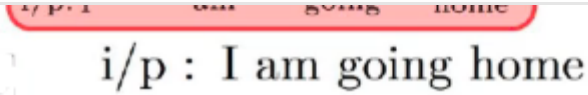
So, the output **yt**(at t'th time step) is a function of all the input words(**x1, x2, x3, ….., xt**) and this is true as all these input words (**x1, x2, x3, ….., xt**) are used to compute the vector **ht**; **ht** is then being used as **s0** for decoder and since it's been used as **s0** it contributes to **s1, s2, s3 and so on** and so it contributes throughout and hence at every



Parameters:

- **Parameters:** U_{dec}, V, W_{dec}, U_{enc}, W_{enc}, b





Loss Function:

At every time step, we want to maximize the probability of the right word(so we minimize the negative of the log of the probability, log is for numerical stability because the probabilities could be very very small and if we take log then it becomes more stable number compared to the small number that we have and floating-point precision does not become so much of an issue or in other words we just use the Cross-Entropy Loss function).

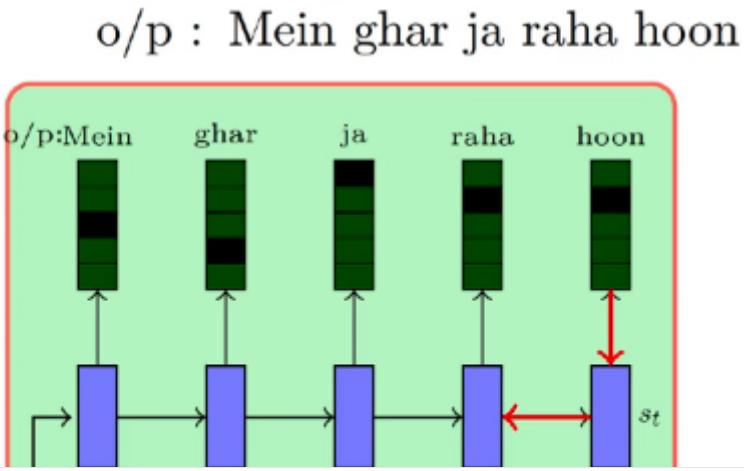
So, the overall loss is the sum of the cross-entropy for all the '**L**' time steps(where **L** is the number of words at the output).

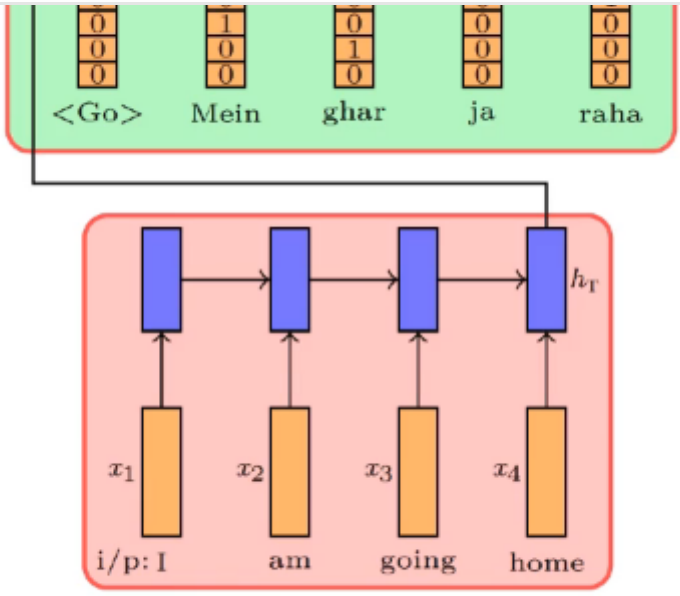
- **Loss:**

$$\mathscr{L}(\theta) = \sum_{i=1}^{t} \mathscr{L}_t(\theta) = -\sum_{t=1}^{t} \log P(y_t = \ell_t | y_1^{t-1}, x)$$

Algorithm:

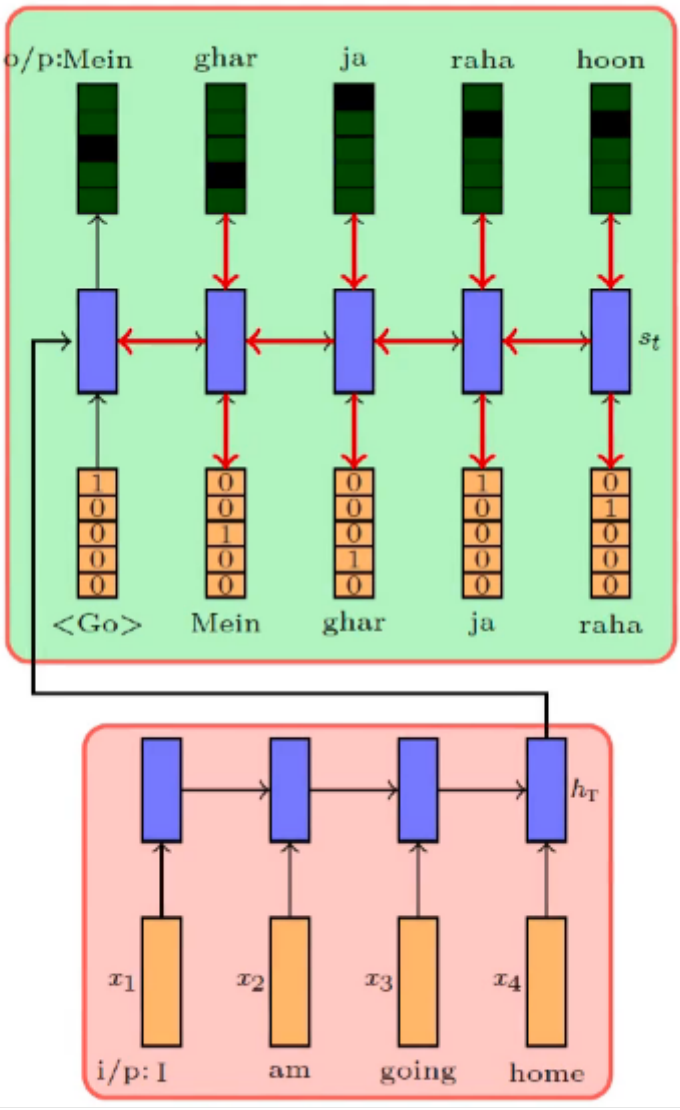
Since this is an RNN, so we use the Training Algorithm as Gradient Descent with backpropagation. We backpropagate the loss all the way back up to the encoder parameters as well(see red arrows in the image below):

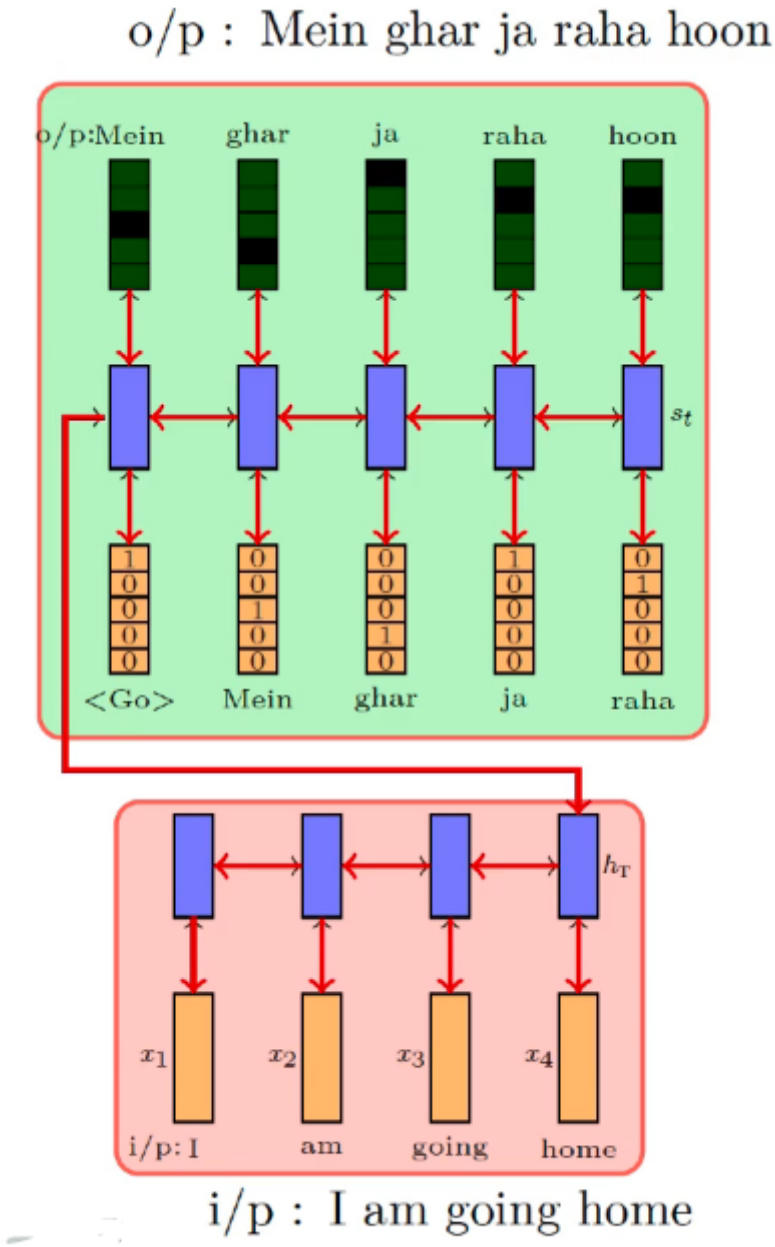




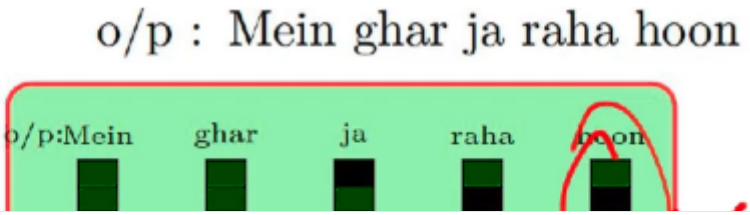
i/p : I am going home

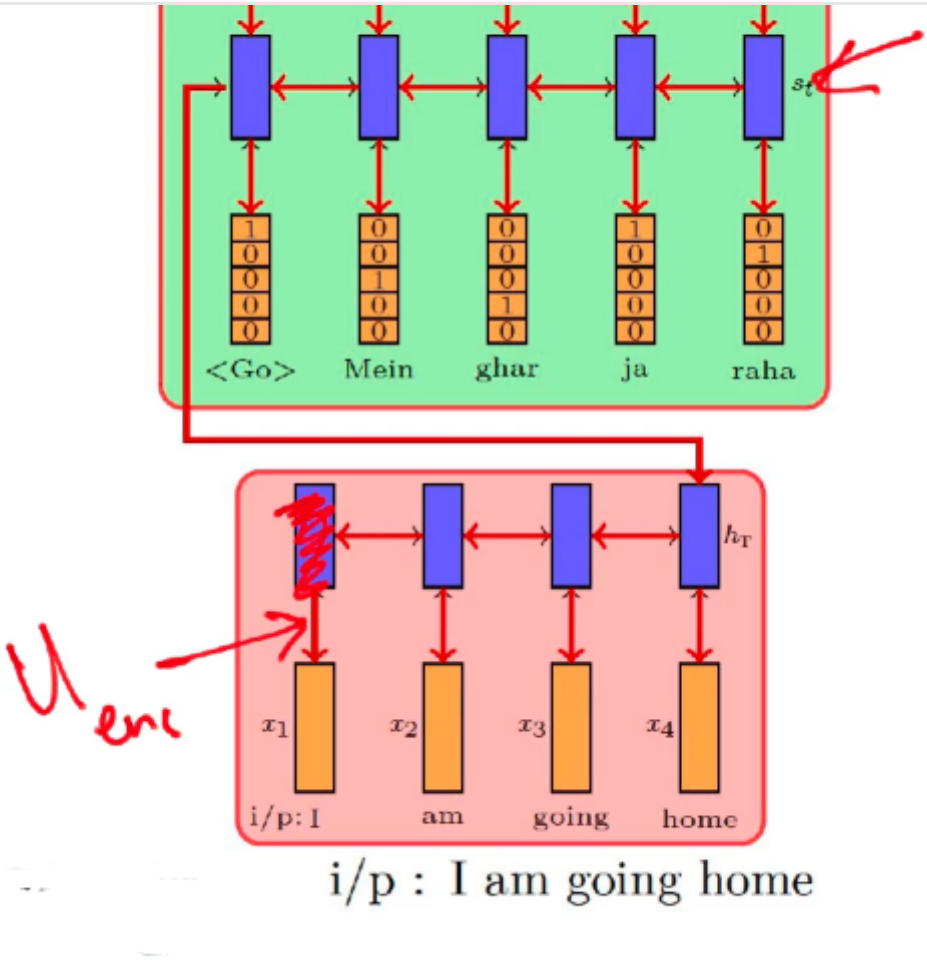
o/p : Mein ghar ja raha hoon





And the reason we backpropagate till encoder parameters is that as it is possible that we get the last word wrong because the **U**(encoder) was not right enough to generate good representation for **h1**(hidden state at **1st time step** for encoder); hence the next hidden state was wrong and so on and hence the last blue vector(**st** in decoder) is wrong which generates the last word incorrectly.

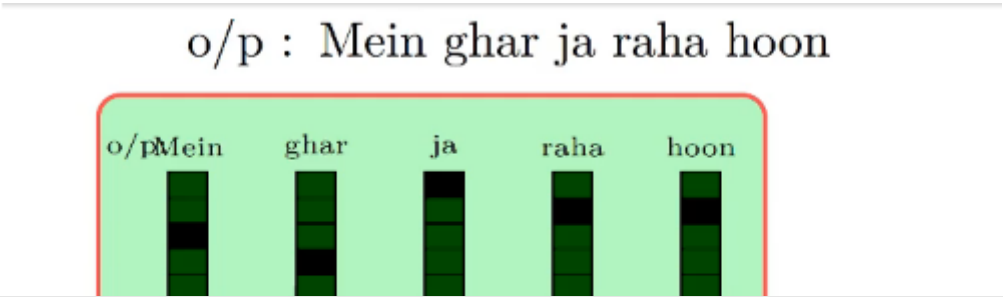


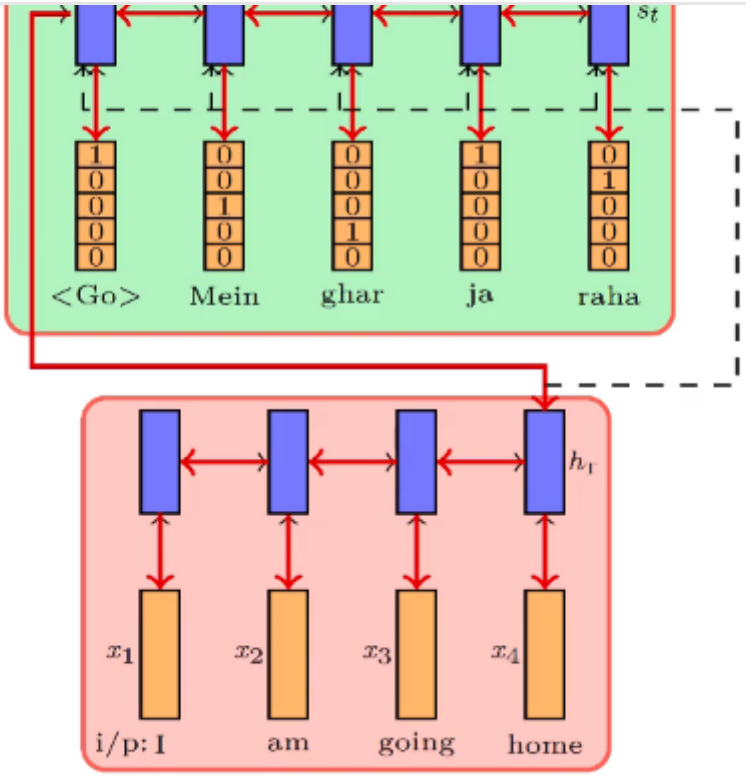


And from the final output to the input is just a continuous chain so we can easily compute the derivative using the chain rule however as this is a long chain, there might be the issue of vanishing and exploding gradients and to avoid that we can also try LSTM instead of RNN.

Model Option 2:

Instead of feeding the final encoded input information at the starting as **s0**(**s0** of the decoder), we can feed this information at every time step(just like we discussed in the Model Option 2 of the Image Captioning Task)





And the equation for computing **st** would change slightly:

- **Task:** Machine translation
- **Data:** $\{x_i = source_i,\ y_i = target_i\}_{i=1}^{N}$
- **Model (Option 2):**
  - **Encoder:**

$$h_t = RNN(h_{t-1}, x_{it})$$

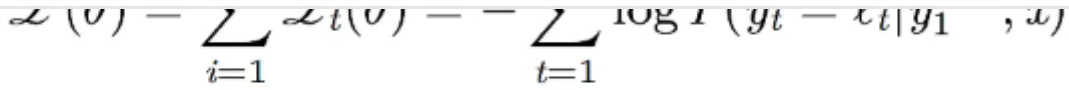
  - **Decoder:**

$$s_0 = h_T \quad (T \text{ is length of input})$$

$$s_t = RNN(s_{t-1}, [h_T, e(\hat{y}_{t-1})])$$

$$P(y_t | y_1^{t-1}, x) = softmax(Vs_t + b)$$

- **Parameters:** $U_{dec},\ V,\ W_{dec},\ U_{enc},\ W_{enc},\ b$



- **Algorithm:** Gradient descent with backpropagation

Earlier we were using the embedding of the previous words but now along with that, we are also concatenating **ht**(highlighted in the above image).

Encoder-Decoder Model for Transliteration:

Task: Our task is Transliteration where given a sequence of characters in one language we want to generate a sequence of characters in another language. And again just as in the case of Translation, it is not a Sequence Labeling Problem because we can't generate a character for every English character. So, this is again a Sequence to Sequence Generation problem.

The solution is exactly the same as that for Translation; the key difference here is going to be that if we look at translation the vocabulary and hence the one-hot vectors that we used was very large and since this is the case of transliteration the number of characters is going to be just 26(in case of English) and in case of Hindi it would be near about 40 or so.

- **Task:** Transliteration

Data: We would need parallel data of the form where we are given an English word and its Hindi transliteration and we need many such word pairs as opposed to sentence pairs in the translation case.

- **Data:** $\{x_i = srcword_i,\ y_i = tgtword_i\}_{i=1}^{N}$

Model:

Option 1 is the same as in the case of Translation case. We treat each character as an input, so we represent each character as one-hot vector and since there are 26

vector and the previous state vector. So, the last hidden state representation of the encoder would have all the input information in the encoded form. Encoder could be RNN or LSTM.

So, whatever we have generated from the last time step of encoder, we feed in to the decoder and then we would generate one character at a time and the way we do this is again we have an RNN which would compute the hidden state based on the previous hidden state and it would also take the one-hot encoding of the character which was predicted at the previous time step and once we have this we can compute the hidden state and from that we can predict the output using a softmax function and this output **y** is a continuous function of the sequence of the input that was given to us.

- **Model (Option 1):**
  - **Encoder:**

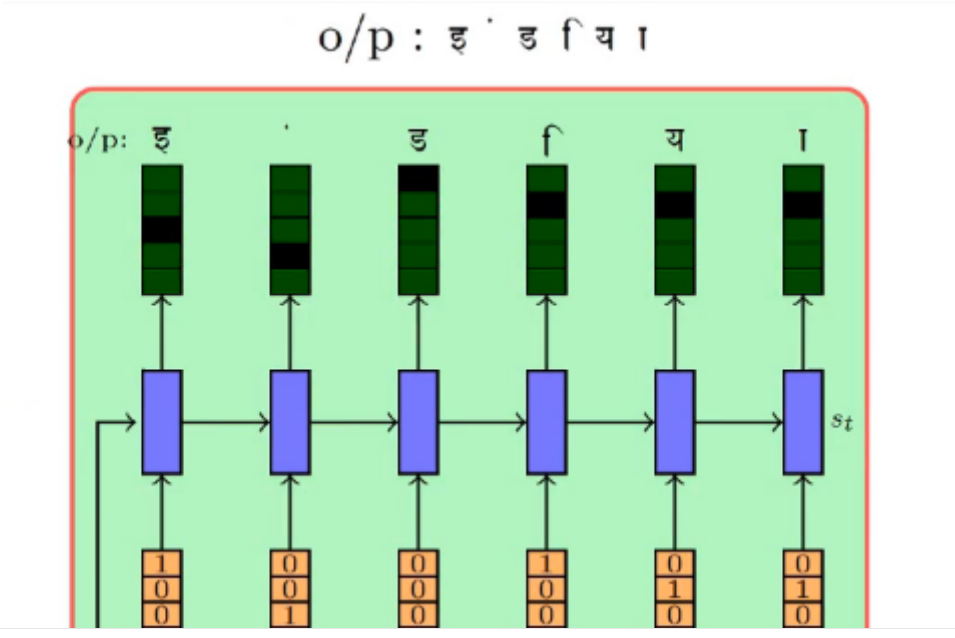
$$h_t = RNN(h_{t-1}, x_{it})$$

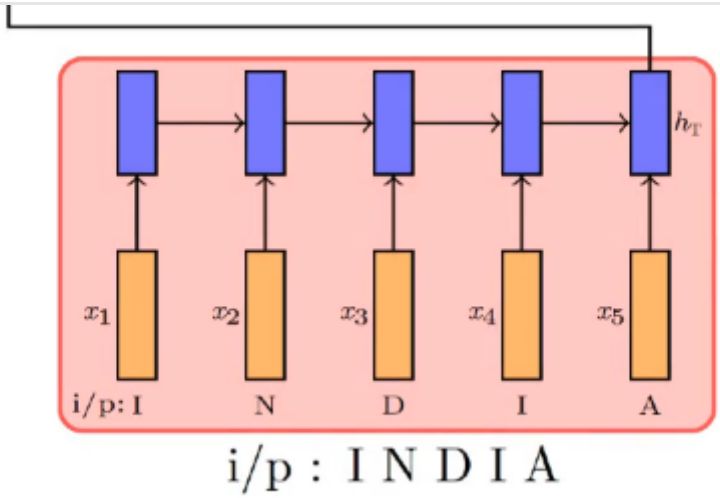
  - **Decoder:**

$$s_0 = h_T \quad (T \text{ is length of input})$$

$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$

$$P(y_t | y_1^{t-1}, x) = softmax(Vs_t + b)$$





- **Parameters:** $U_{dec},\ V,\ W_{dec},\ U_{enc},\ W_{enc}, b$

Loss Function would be the sum of the Cross-Entropy Loss or the negative log-likelihood of the data where **lt** is the true character at that time step

- **Loss:**

$$\mathscr{L}(\theta) = \sum_{i=1}^{T} \mathscr{L}_t(\theta) = -\sum_{t=1}^{T} \log P(y_t = \ell_t | y_1^{t-1}, x)$$

So, at this time step, it should have generated **lt**, if that is not being generated then we want to maximize the probability of **lt** and we give feedback to the model to do that and this feedback would flow back through time and it would go all the way back to the input also.

- **Loss:**

$$\mathscr{L}(\theta) = \sum_{i=1}^{T} \mathscr{L}_t(\theta) = -\sum_{t=1}^{T} \log P(y_t = \ell_t | y_1^{t-1}, x)$$

Option 1:

- **Task:** Transliteration
- **Data:** $\{x_i = srcword_i,\ y_i = tgtword_i\}_{i=1}^{N}$
- **Model (Option 1):**
  - **Encoder:**

$$h_t = RNN(h_{t-1}, x_{it})$$

  - **Decoder:**

$$s_0 = h_T \quad (T \text{ is length of input})$$
$$s_t = RNN(s_{t-1}, e(\hat{y}_{t-1}))$$
$$P(y_t|y_1^{t-1}, x) = softmax(Vs_t + b)$$

- **Parameters:** $U_{dec},\ V,\ W_{dec},\ U_{enc},\ W_{enc}, b$
- **Loss:**

$$\mathscr{L}(\theta) = \sum_{i=1}^{T} \mathscr{L}_t(\theta) = -\sum_{t=1}^{T} \log P(y_t = \ell_t|y_1^{t-1}, x)$$

- **Algorithm:** Gradient descent with backpropagation

Option 2:

At every time step, we feed in the encoding of all the input words also in addition to the current input and all the previous time steps.

- **Task:** Transliteration
- **Data:** $\{x_i = srcword_i,\ y_i = tgtword_i\}_{i=1}^{N}$

- **Encoder:**

$$h_t = RNN(h_{t-1}, x_{it})$$

- **Decoder:**

$$s_0 = h_T \quad (T \text{ is length of input})$$

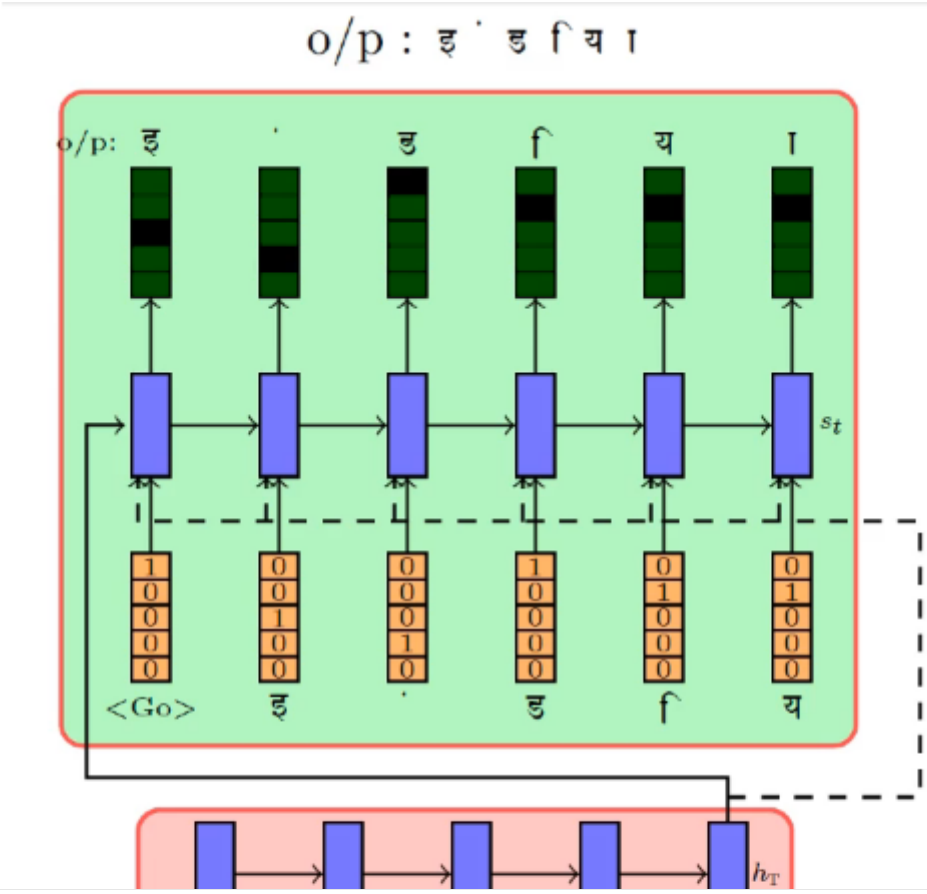
$$s_t = RNN(s_{t-1}, [e(\hat{y}_{t-1}), h_T])$$

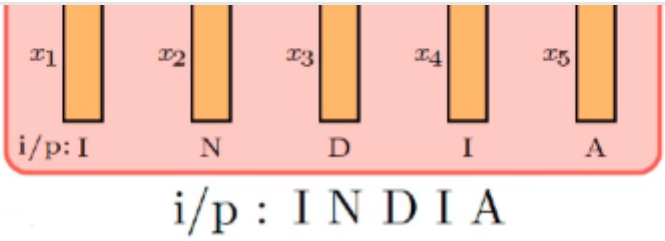
$$P(y_t | y_1^{t-1}, x) = softmax(Vs_t + b)$$

- **Parameters:** U_{dec}, V, W_{dec}, U_{enc}, W_{enc}, b
- **Loss:**

$$\mathscr{L}(\theta) = \sum_{i=1}^{T} \mathscr{L}_t(\theta) = -\sum_{t=1}^{T} \log P(y_t = \ell_t | y_1^{t-1}, x)$$

- **Algorithm:** Gradient descent with backpropagation





i/p : I N D I A

All the images used in this article is taken from the content covered in the Vanishing and Exploding module of the Deep Learning Course on the site: padhai.onefourthlabs.in

Rnn Deep Learning Image Captioning Machine Translation Padhai